Rule Based Programming with Constraints & Strategies

Hubert Dubois, Hélène Kirchner

Rule Based Programming with Constraints and Strategies

Hubert Dubois and Hélène Kirchner
LORIA-CNRS & UHP Nancy 1
BP 239 54506 Vandœuvre-lès-Nancy Cedex, France
e-mail: Hubert.Dubois@loria.fr,Helene.Kirchner@loria.fr

Abstract

We present a framework for Rule Based Programming with Constraints and Strategies. It is based on an extension of the ELAN language, that provides an environment for specifying and prototyping deduction systems. The existence of strategies provides the user with the possibility to make choices, to act upon them, and to retract if needed using bachtracking. To illustrate the framework, we formalise a planning problem, namely a controller for printing tasks, that shows how to combine rules, strategies and constraint solving on finite domains.
Keywords: Rule based programming, rewrite rules, control, strategy, planning problems.

1 Introduction

Forward chaining rule-based systems (RBS for short) have been widely used in artificial intelligence and expert systems. A RBS is basically a set of condition-action rules: when the conditions are matched by some facts, the actions are performed. This formalism is well-suited for reactive and control tasks. However in these systems, it is difficult to make use of disjunctive information and to reason about choices. A framework called CRP for Constraint Rule-based Programming, has been recently proposed by Liu, Jaffar and Yap in [8].It extends the basic operational model of RBS to include the notion of constraint solving. We started from a similar idea, but build up on the ELAN system [3], that provides an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. ELAN offers a natural and simple logical framework that supports the design of theorem provers, logic programming languages, constraint solvers and decision procedures and offers a modular framework for studying their combination. Among many applications developed in ELAN, the system COLETTE provides various techniques for solving constraint satisfaction problems [4], uniformly based on rewrite rules and strategies.

Our proposition improves ELAN by allowing in the language structured objects and constraints into rewrite rules. It improves CRP by adding strategies, i.e. declarative control on rules. We thus provide in a same context rules, constraints and strategies.

The proposed framework has the following components:

- A configuration that is a multiset of structured objects representing the current state of the system. Objects may contain constrained variables, i.e. variables whose possible values are related to a constraint involving several variables.

- A set of conditional rules that transform the configuration and describe the dynamic aspect of the system. A rule applies if objects occurring in the left-hand side match corresponding objects in the configuration, and if the conditions specified in the rule are satisfied. Rules may involve constraints, i.e. relations between values of attributes in objects occurring in the left-hand side. Rules have labels that allows calling them in controlling their application through strategies.
- Control is defined by strategies and allows backtracking, choices and search.
- A constraint store, that represents the current constraints in the system. The constraint store can also be modelled as a structured object or a configuration. Other rules express transformations of the constraint store when a constraint is reduced to a simpler form. Strategies express how checking satisfiability of constraints or enumerating solutions

To illustrate the approach, we consider on an example, the design of a print controller, inspired from [1]. The goal of the print controller is to help the user to decompose a complex task into a sequence of simple actions, to determine which sets of actions are capable of achieving a given complex print task, and to generate at any step the space of possible decompositions, in order to help the user to take decisions. Since all print tasks must be achieved before a given deadline, constraints associated to this problem are time constraints involving begin and end times of each task.

In Section 2, we go through the different components of the framework and introduce the corresponding language constructions to write programs in this framework. In Section 3, we explain how the framework is implemented in ELAN. We define a translation from our programs to ELAN programs, in order to reuse ELAN execution mechanism for programs evaluation. This translation has the advantage to provide both a logical and operational semantics in rewriting logic for our framework. In Section 4, we propose as conclusion further research directions.

2 Description of the language

In order to write programs in the proposed framework, we need a language to formalise the different components, which are the structured objects (Section 2.1), the constraint store (Section 2.2), the rules (Section 2.3), and the strategies (Section 2.4).

2.1 Structured objects

Each structured object has a name, a type whose purpose is to classify these objects, and a description given by set of pairs (attribute, value). A structured object is represented as an expression

$$|Name \ : \ Class \ :: \ a_1 = v_1 \ , \ \ldots \ , \ a_{N_{Class}} = v_{N_{Class}}|$$

where $Class$ identifies the class (type) of the object named by the identifier $Name$, and $(a_1 = v_1, \ldots, a_{N_{Class}} = v_{N_{Class}})$ is the list of pairs (attribute, value) that caracterises this object. The order of attributes in the list is irrelevant. All objects in a same class have the same number of attributes N_{Class}.

Operations provided on all classes of structured objects are:

- return the value associated to an attribute by the operator $\_\cdot\_$ where $\_$ denotes a place-holder for argument. For instance, $Name \cdot a$ returns the value v associated to the attribute a of the object named $Name$.

- modify the value associated to an attribute by the operator $\_[\_ \leftarrow \_]$. For instance, associating the value v to the attribute a of an object $Name$ is denoted by $Name[a \leftarrow v]$.

Each class of objects may also contain additional operations that modify the objects (i.e. the values of its attributes), and functions that perform additional computations but do not change the objects of the class.

Example 1 *In the formalisation of the print controller, there are simple print tasks of class Action, and complex print tasks of class Task. A simple print task consists of printing once a document. The duration associated to this task is fixed and it cannot be interrupted. A complex print task consists of loading in memory the document to print, of printing it several times and when finished, of freeing the allocated memory space. This complex print task can be interrupted but, in this case, the allocated memory space must be released and when starting again this job, the document has to be reloaded.*

Tasks (objects of class Task) have five attributes giving respectively: their name, the number of impressions that the task must manage, the status for the decomposition of this task (that can be either all or split), a begin time and an end time.

Actions (objects of class Action) have three attributes giving respectively: their name, the begin time of the action, and their end time.

2.2 The constraint store

We focus in this paper on constraints on finite domains, useful for the kind of applications we consider. But in general, there is no restriction on the kind of constraints handled in the constraint store.

We consider here Constraint Satisfaction Problems (CSP for short). A CSP is defined by a set of variables, their respective domains and a set of constraints built as disjunctions and conjunctions of equations, inequations and disequations on arithmetic expressions with addition and substraction on finite domains.

The CSP is a structure with two fields:

- $Varlist$ giving the list of membership constraints of the form $(X \in^? D)$ where X is a variable and D its domain,
- $Conset$ giving the set of constraints.

Example 2 *Considering three variables V_1, V_2 and V_3 taking their values in the finite domain $[0,..,10]$ and the following constraint $V_1+V_2 =^? V_3 \wedge V_1 \geq^? 4 \wedge V_2 \leq^? 8 \wedge V_3 >^? 2$. The CSP corresponding to this problem is composed of two fields:*

$$VarList = V_1 \in^? [0,..,10], V_2 \in^? [0,..,10], V_3 \in^? [0,..,10]$$

$$Conset = V_1 + V_2 =^? V_3 \wedge V_1 \geq^? 4 \wedge V_2 \leq^? 8 \wedge V_3 >^? 2$$

In order to formalize constraint solving and satisfiability checking, the structure of the constraint store may be more complex. For instance, the attribute $Conset$ can be split into four sets: a list of equality constraints of the form $(X =^? V)$ where V is an arithmetic expression, a list of conjunctive constraints and a list of disjunctive constraints. The last component stores intermediate results.

Operations available on the constraint store are: get the list of variables, get the current domain of a variable, check if a variable has a unique value in its domain, check if a variable has an empty domain (the CSP is unsatifiable), add a new constraint, check the satisfiability of the CSP, solve a CSP.

The constraint solver for constraints on finite domains is based on local propagation techniques and enumeration with backtracking. The formalism of rules controlled by strategies gives a lot of flexibility to design appropriate strategies to solve constraints, to check consistency or to compute all solutions. We always assume that there exists at least two functions: `Sat` to check consistency of the CSP and `Solve` to enumerate its solutions. These functions can be implemented using different strategies, as studied for instance in [4, 5].

2.3 Rules

Rules describe changes of the configuration and are of the form:

$$[lab]\ \ O_1 \ldots O_k\ \Rightarrow\ O'_1 \ldots O'_m\ \ [\textbf{if}\ t \mid \textbf{where}\ l]^*\ \|\ c$$

where each $O_1, \ldots, O_k, O'_1, \ldots, O'_m$ are structured objects, t is a boolean term called condition, l a local assignment to define auxiliary variables, and c is a constraint, given here as a CSP. This rule can have the label $[lab]$, or no label which is denoted [].

We denote O_1 ... O_k the objects appearing in the left hand side (lhs for short) of the rule. In the right hand side (rhs) of a rule, we can find objects that appear also in the lhs and which can be unchanged or modified, and new objects created by this rule.

A rule is candidate if its lhs matches a subset of structured objects in the configuration. Application of this rule succeeds only if the tests **if** t are valid and the local assignments **where** l succeed. The configuration is then updated by modifying instantiated objects ocurring in the lhs according to their instances in the rhs, and adding instances of new objects ocurring in the rhs but not in the lhs. Instances of objects occurring in the lhs but not in the rhs are deleted in the configuration.

In the constraint part c of rules that defines the interface between the rules and the constraint solver, we give two components: a domain initialization for each new variable of the contraint and the new constraint to be added to the store.

Note that constraints and constraint variables are restricted to appear in the right-hand sides of rules because they change the constraint store. This restriction prevents the interference between matching and constraint solving. A similar restriction is done in CRP.

Example 3 *In the print controller example, let us consider the problem of finding possible schedules for executing on the printer a simple print task and a complex one composed of N print tasks. The different possibilities are either to split the complex print task or to perform it in one step before or after the simple print task. Note that the purpose is not to minimize the execution time for these requests, but to find all possible schedules within the given deadline.*

Five rules decompose complex tasks into other complex tasks or into primitive actions.

The first rule specifies the different tasks to be scheduled: here the task Main (denoted M) is composed of a complex task FormPrint (FP) to perform, and of a primitive action SimplePrint (SP) of duration 2 time units. Constraint variables are put in attributes B and E of objects $O2$ and $O3$ and a constraint involving these variables is sent to the constraint store. Each variable is initialized with a domain D equal to $[0, .., T]$ where T is the end time wanted for the main task. The constraint put on begin and end times of the different tasks must express that either SP precedes FP, or FP precedes SP, or FP begins before SP and ends after SP. In a more formal way: $FP \cdot B =^? SP \cdot E \ \vee\ SP \cdot B =^? FP \cdot E \ \vee\ (SP \cdot B \geq^? FP \cdot B \ \wedge\ SP \cdot E \leq^? FP \cdot E)$.

Instead of considering this disjunctive constraint, we choose here to write three rules labelled Main1, Main2 *and* Main3 *each corresponding to a disjuntion. We will see later how strategies can insure to explore the three possibilities.*

The two following rules are for decomposing the complex task FormPrint (FP). There are two possible choices here: either we do not decompose this task, but we load (noted L - of duration 1 time unit) the document, multi-print it N times, while we assure that the document is kept in memory during all these operations (this is the primitive action FormKeep (noted FK)). This choice involves that the SP action cannot happen during this task. This corresponds to the F1 decomposition. Or we decide to decompose the complex task into two parts. The initial number N of documents to print is split into $N1$ and $N2$ such that $N1 + N2 = N$. This is achieved by the non-deterministic function `Split`. *This corresponds to the F2 decomposition. Moreover, when creating a new FormPrint task, its status is given by a non-deterministic choice expressed by the strategy* `ChooseStatus`.

The two last rules laballed MP1 and MP2 decompose the complex task MultiPrint (MP). It simply consists, for printing a document N times, in printing it once (the duration of a print is 1 time unit) and then $N - 1$ times.

[Main1] $|O1 \ : \ Task \ :: \ Name = Main| \Rightarrow$
$|O2 : Task :: Name = FormPrint, Status = all, NumI = O1.NumI - 1, B = V\_1, E = V\_2|$
$|O3 \ : \ Action \ :: \ Name = SimplePrint \ , \ B = V\_3 \ , \ E = V\_4|$
$\| \ \{V\_1 \in^? D, V\_2 \in^? D, V\_3 \in^? D, V\_4 \in^? D\}, \ V\_1 \geq^? O1.B \ \wedge \ V\_3 =^? O1.B \ \wedge$
$V\_2 \leq^? O1.E \ \wedge \ V\_4 \leq^? O1.E \ \wedge \ V\_4 =^? V\_1 \ \wedge \ V\_4 =^? V\_3(+)2$

[Main2] $| \ O1 \ : \ Task \ :: \ Name = Main| \Rightarrow$
$|O2 : Task :: Name = FormPrint, Status = all, NumI = O1.NumI - 1, B = V\_1, E = V\_2|$
$|O3 \ : \ Action \ :: \ Name = SimplePrint \ , \ B = V\_3 \ , \ E = V\_4|$
$\| \ \{V\_1 \in^? D, V\_2 \in^? D, V\_3 \in^? D, V\_4 \in^? D\} \ , \ V\_1 =^? O1.B \ \wedge \ V\_1 >^? O1.B \ \wedge$
$V\_2 \leq^? O1.E \ \wedge \ V\_4 \leq^? O1.E \ \wedge \ V\_4 =^? V\_3(+)2 \ \wedge \ V\_2 =^? V\_3$

[Main3] $| \ O1 \ : \ Task \ :: \ Name = Main| \Rightarrow$
$|O2 : Task :: Name = FormPrint, Status = split, NumI = O1.NumI - 1, B = V\_1, E = V\_2|$
$|O3 \ : \ Action \ :: \ Name = SimplePrint \ , \ B = V\_3 \ , \ E = V\_4|$
$\| \ \{V\_1 \in^? D, V\_2 \in^? D, V\_3 \in^? D, V\_4 \in^? D\} \ , \ V\_1 \geq^? O1.B \ \wedge \ V\_3 \geq^? O1.B \ \wedge$
$V\_2 \leq^? O1.E \ \wedge \ V\_4 \leq^? O1.E \ \wedge \ V\_4 =^? V\_3(+)2 \ \wedge \ V\_3 >^? V\_1 \ \wedge \ V\_4 <^? V\_2$

[F1] $| \ O1 \ : \ Task \ :: \ Name = FormPrint \ , \ Status = all \ , \ NumI = N|$
$|O2 \ : \ Action \ :: \ Name = SimplePrint| \ \Rightarrow$
$|O2 \ : \ Action \ :: \ Name = SimplePrint|$
$|O3 \ : \ Action \ :: \ Name = Load \ , \ B = V\_1 \ , \ E = V\_2|$
$|O4 \ : \ Task \ :: \ Name = MultiPrint \ , \ NumI = N \ , \ B = V\_3 \ , \ E = V\_4|$
$|O5 \ : \ Action \ :: \ Name = FormKeep \ , \ B = V\_5 \ , \ E = V\_6|$
$\| \ \{V\_1 \in^? D, V\_2 \in^? D, V\_3 \in^? D, V\_4 \in^? D, V\_5 \in^? D, V\_6 \in^? D\}, \ V\_1 =^? O1.B \ \wedge$
$V\_2 =^? V\_1(+)1 \ \wedge \ V\_3 =^? V\_2 \ \wedge \ V\_4 =^? O1.E \ \wedge \ V\_5 =^? V\_1 \ \wedge \ V\_6 =^? V\_4 \ \wedge$
$O2.B \geq^? V\_4 \ \vee \ O2.E \leq^? V\_1$

[F2] $|O1 : Task :: Name = FormPrint\ ,\ Status = split\ ,\ NumI = N| \Rightarrow$
$|O2 : Task :: Name = FormPrint, NumI = N1, Status = st1, B = V\_1, E = V\_2|$
$|O3 : Task :: Name = FormPrint, NumI = N2, Status = st2, B = V\_3, E = V\_4|$
where $[N1, N2] := () \ split(N)$
where $[st1, st2] := () \ ChooseStatus$
$\| \{V\_1 \in^? D, V\_2 \in^? D, V\_3 \in^? D, V\_4 \in^? D\},\ V\_1 =^? O1.B \ \wedge\ V\_3 >^? O1.B \ \wedge$
$V\_2 \leq^? V\_3 \ \wedge\ O1.E =^? V\_4$

[MP1] $|O1 : Task :: Name = MultiPrint\ ,\ NumI = 1| \Rightarrow$
$|O2 : Action :: Name = Print\ ,\ B = V\_1\ ,\ E = V\_2|$
$\| \{V\_1 \in^? D, V\_2 \in^? D\}\ ,\ V\_1 =^? O1.B \ \wedge\ V\_2 =^? V\_1(+)1 \ \wedge\ O1.E =^? V\_2$

[MP2] $|O1 : Task :: Name = MultiPrint\ ,\ NumI = N| \Rightarrow$
$|O2 : Action :: Name = Print\ ,\ B = V\_1\ ,\ E = V\_2|$
$|O3 : Task :: Name = MultiPrint\ ,\ NumI = N - 1\ ,\ B = V\_3\ ,\ E = V\_4|$
if $N > 1$
$\| \{V\_1 \in^? D, V\_2 \in^? D, V\_3 \in^? D, V\_4 \in^? D\},\ V\_1 =^? O1.B \ \wedge$
$V\_2 =^? V\_1(+)1 \ \wedge\ V\_2 =^? V\_3 \ \wedge\ O1.E =^? V\_4$

Figure 1: Rules for the print controller.

2.4 Strategies

In general, the application of a rule on the configuration may return several results, in case of several objects or mutisets of objects match the left-hand side. These sets of results are handled through a backtracking mechanism.

In order to take into account non-determinism and sets of results, and to control rule application, the concept of strategy is introduced: a strategy is a function which, when applied to a configuration, returns a set of possible configurations. The strategy fails if the set is empty. To define strategies, the following strategy constructors are provided:

- A labelled rule is a primal strategy. The result of applying a rule labelled lab on a configuration returns a set of configurations. This primal strategy fails if the set of results is empty.
- Two strategies can be concatenated by the symbol ";", i.e. the second strategy is applied on all results of the first one. $S_1; S_2$ denotes the sequential composition of the two strategies. It fails if either S_1 fails or S_2 fails. Its results are all results of S_1 on which S_2 is applied and gives some results.
- $\mathsf{dk}(S_1, \ldots, S_n)$ chooses all strategies given in the list of arguments and for each of them returns all its results. This set of results may be empty, in which case the strategy fails.
- $\mathsf{first}(S_1, \ldots, S_n)$ chooses in the list the first strategy S_i that does not fail, and returns all its results. This strategy may return more than one result, or fails if all sub-strategies S_i fail.

- first_one$(S_1, \ldots, S_n)$ chooses in the list the first strategy S_i that does not fail, and returns its first result. This strategy returns at most one result or fails if all sub-strategies fail.
- The strategy id is the identity that does nothing but never fails.
- fail is the strategy that always fails and never gives any result.
- repeat*(S) applies repeatedly the strategy S until it fails and returns the results of the last unfailing application. This strategy can never fail (zero application of S is possible) and may return more than one result.
- The strategy iterate*(S) is similar to repeat*(S) but returns all intermediate results of repeated applications.

The easiest way to build a strategy is to use the strategy constructors to build strategy terms and to define a new constant operator that denotes this (more or less complex) strategy expression. This gives rise to a first class of strategies called elementary strategies. Elementary strategies are defined by unlabelled rules of the form [] $S \rightarrow$ *strat*, where S is a constant strategy operator and *strat* a term built on predefined strategy constructors and rule labels, but that does not involve S. The application of a strategy S on a term t is denoted $(S)t$. Recursive and parameterised strategies may also be defined and more examples can be found in [2].

Example 4 *The three rewrite rules labelled by* `Main1`, `Main2` *and* `Main3` *have to be applied non-deterministically since they correspond to three simultaneous cases, each one caracterised by a different constraint. Note that this is a way to deal with disjunctive constraints.*

The strategy `Main` *is defined as a non-determistic choice of these three* ELAN *rules.*

The FormPrint complex task is also defined by two rules in ELAN *labelled* `F1` *for FP with status all,* `F2` *for FP with status split, again each one corresponding to a disjunct in the constraint. The* `Form` *associated strategy is again a non-determistic choice of these* ELAN *rules.*

For the MultiPrint task, we have also two rules in ELAN *labelled* `MP1` *and* `MP2` *and the associated strategy is called* `Multi`.

These three strategies are defined as follows:

```
[] Main    => dk (Main1 , Main2 , Main3)
end
[] Form    => dk (F1 , F2)
end
[] Multi   => first (MP1 , MP2)
end
```

Note that the rules used here do not check satisfiability of the constraint store. We can easily design another process that would check at each step a satisfiability check of the constraint store. Assume that we have defined a strategy `Sat` *on the constraint store that checks whether the constraint store admits at least a solution. Then we can modify the three strategies above by including* `Sat` *whenever wanted. For instance to impose at each step a satisfiability check.*

```
[] MainSat    => dk (Main1;Sat , Main2;Sat , Main3;Sat)
end
[] FormSat    => dk (F1;Sat , F2;Sat)
```

```
end
[] MultiSat   => first (MP1;Sat , MP2;Sat)
end
```

3 Implementation in ELAN

3.1 The data structure

The state of a concurrent object system, i.e. the configuration, is a multiset of objects built with the constructor $\{\_, \ldots, \_\}$. Configurations are built up by a binary multiset union operator which is associative and commutative (AC for short). Objects are singleton multiset configurations. The empty configuration *nobj* is an identity for multiset union.

In ELAN, $[\_, \ldots, \_]$ is tuple constructor with a flexible arity, allowing the construction of pairs $[\_, \_]$, 3-tuples $[\_, \_, \_]$, etc... So an object $|\_ : \_ :: \_|$ is represented as a term with root operator $[\_, \_, \_]$. The third argument, i.e. the list of attributes, is implemented as a multiset of pairs $[a_i, v_i]$ and *natt* is the empty list. The order of (attribute, value) pairs is irrelevant, due to properties of associativity and commutativity of union.

The constraint store is also represented as a term corresponding to the structured term presented in Section 2.2. The first argument is a list of domain definition for each variable and the second one is the constraint. It is represented by a pair.

We can define operators on the constraint store:

- $\_.VarList$ returns the list of variables, i.e. the first element of the pair composing the constraint store. The argument is a constraint store.
- $\_.Conset$ returns the constraint, i.e. the second element of the pair composing the constraint store. The argument is also a constraint store.
- $\_.Max$ returns the number of variables used in the constraint store. This operator is useful to create new variables and to guarantee their unicity.
- $\_\uplus[\_, \_]$ constructs a new constraint store built from an initial constraint store (the first argument) updated with a new CSP composed by a list of new variables (the second argument) and a new constraint (the last argument).

Having defined how the configuration and the constraint store are represented, we can introduce a sort `world` and a constructor **_with_** for this sort, taking as first component a configuration, and as second component a constraint store.

3.2 Constraint solving and satisfiability checking

For solving constraint satisfaction problems in our example, we use the COLETTE system defined by C.Castro [5].

The formalism of CSP used by COLETTE is quite close to the formalism used in this approach.

When giving a CSP to the COLETTE solver, we have to initialize the five components of the constraint store from the list of variables and the constraint. Then, the constraint store is simplified (i.e. domains of variables are reduced by a local consistency technique). COLETTE also provides a library of strategies for solving constraint satisfaction problems. Among them let us mention: the Forward Checking strategy and the Full Look Ahead one.

We give below some results obtained with the Forward Checking strategy with an enumeration of the results from left to right. This strategy is called in COLETTE `FCFirstToLast`. We have used this strategy in two ways. The first one is the entire solving of the CSP and the enumeration of all solutions in order to produce the strategy `Solve`. The second way just consists of stopping the enumeration as soon as the first solution is found, in order to produce the strategy `Sat`. We give here the strategy `Sat` obtained by adapting the strategy `FCFirstToLast` described in [5].

```
[] Sat =>
repeat* (
        dk (iterate* (EliminateFirstValueOfDomain)) ;
        first one (InstantiateFirstValueOfDomain) ;
        first one (ExtractConstraintsOnEqualityVar, id) ;
        first one (Elimination , id) ;
        first one (LocalConsistencyInEC , id)
        )
first one (GetSolutionCSP2)
end
```

3.3 Translation of rules

ELAN provides a very general form of a rule, defined as follows:

$$[\ell]\ \ l \to r \ \textbf{where}\ p_1 := (S_1)c_1\ \ \ldots\ \ \textbf{where}\ p_n := (S_n)c_n$$

Defining $\mathcal{T}(\mathcal{F},\mathcal{X})$ as the set of *terms* built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables and $\mathcal{V}ar(t)$ as the set of variables occurring in a term t, variables and term occurings in the rule must satisfy the following conditions:

- $l, r, p_1, \ldots, p_n, c_1, \ldots, c_n \in \mathcal{T}(\mathcal{F},\mathcal{X})$,
- $\forall i, i = 1, \ldots, n$, p_i and $(S_i)c_i$ have the same sort.
- $\forall i, i = 1, \ldots, n$, $\mathcal{V}ar(p_i) \cap (\mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})) = \emptyset$,
- $\forall i, i = 1, \ldots, n$, $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})$,
- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_n)$,
- $S_1, \ldots, S_n$ are strategy terms and $(\_)\_$ is the application operator of strategies on terms.

Let us now explain how the rules are translated to ELAN rules. Concerning the structured objects component, the translation is similar to the translation of object-oriented modules in FullMaude into system modules in Maude, described in [6].

The rule

$$[lab] \quad O_1 \ldots O_k \Rightarrow O'_1 \ldots O'_m \quad [\textbf{if}\ t \mid \textbf{where}\ l]^* \parallel c$$

is transformed into an ELAN rule given in Figure 2.

Unspecified arguments of AC operators are captured by new variables added in the left-hand sides of rewrite rules. Let Z be the variable of sort Configuration which captures the other objects of the configuration; let At_i the variable capturing the (attribute, value) pairs unspecified by the rule. This corresponds to the lhs of the rule shown in Fig. 2, line 1.

In this translation, we decompose the m objects in the rhs into three sets:

1 $[Label]\ \{[X_i, Class_i, [a_1, v_1] \ldots [a_{n_i}, v_{n_i}]\ At_i]\}_{i=[1,\ldots,k]}\ Z$ **with** C
$\Rightarrow \{O_i\}_{i\in[1,\ldots,k]}\ X'_1 \ldots X'_m\ Z$ **with** $C1$

2 **where** $O_1 := ()\ [X_1, Class_1, [a_1, v_1] \ldots [a_{n_1}, v_{n_1}]\ At_1]$

3 ...

4 **where** $O_k := ()\ [X_k, Class_k, [a_1, v_1] \ldots [a_{n_k}, v_{n_k}]\ At_k]$

5 **where** $X'_1 := ()\ O_1[\{Att_j \leftarrow Val_j\}_{j=[1,\ldots,n_1]}]$

6 ...

7 **where** $X'_{m_1} := ()\ O_{m_1}[\{Att_j \leftarrow Val_j\}_{j=[1,\ldots,n_{m_1}]}]$

8 **where** $X'_{k+1} := ()\ [O(k+1), Class_{k+1}, \{[a_j, v_j]\}_{j=[1,\ldots,n_{k+1}]}$
$\{[a_j, V\_C.Max + j)]\}_{j=[n_{k+1}+1,\ldots,N_{CVar_{k+1}}]}\ \{[a_j, \perp],\}_{j=[N_{CVar_{k+1}}+1,\ldots,N_{k+1}]}\ natt]$

9 ...

10 **where** $X'_m := ()\ [O(m), Class_m, \{[a_j, v_j]\}_{j=[1,\ldots,n_m]}$
$\{[a_j, V\_C.Max + j)]\}_{j=[n_m+1,\ldots,N_{CVar_m}]}\ \{[a_j, \perp],\}_{j=[N_{CVar_m}+1,\ldots,N_m]}\ natt]$

11 [**if** t | **where** $l]^*$

12 **where** $C1 := ()\ C \uplus [C.VarList, C.Conset]$

Figure 2: Translation of an object-rule.

- persistent objects that are in the lhs and not changed. They corresponds to objects O_i in rhs (line 1): they appears in the lhs under the form $[X_i, Class_i, [a_1, v_1] \ldots [a_{n_i}, v_{n_i}]\ At_i]$ and are denoted using variables O_i (lines 2 to 4).
- modified objects from lhs that are changed by the rule ; they are denoted by new variables X'_1 to X'_{m_1} defined from line 5 to 7.
- new objects that are created by the rule and indexed from $(k+1)$ to m (lines 8-10). Each attribute is initialized either by a value given in the rule, or by a new variable, used in the constraint store and denoted V_i, or by a default value noted $\perp$.

Then, the lines corresponding to the **if** and **where** statements are reproduced. The constraint part is translated by the updating of the constraint store using the operator $\uplus$.

Example 5 *Let us consider the translation of the first rule in the print controller example and given in Figure1:*

[Main1] $|\ O1\ :\ Task\ ::\ Name = Main| \Rightarrow$
$|O2 : Task :: Name = FormPrint, Status = all, NumI = O1.NumI - 1, B = V\_1, E = V\_2|$
$|O3\ :\ Action\ ::\ Name = SimplePrint\ ,\ B = V\_3\ ,\ E = V\_4|$
$\|\ \{V\_1 \in^? D, V\_2 \in^? D, V\_3 \in^? D, V\_4 \in^? D\},\ V\_1 \geq^? O1.B\ \wedge\ V\_3 =^? O1.B\ \wedge$
$V\_2 \leq^? O1.E\ \wedge\ V\_4 \leq^? O1.E\ \wedge\ V\_4 =^? V\_1\ \wedge\ V\_4 =^? V\_3(+)2$

Following the transformation rule given in 2, we get this ELAN *rule with the corresponding variables declaration:*

$Vars$ $X'2, X'3, O1 : Object$
$X_1 : ObjectIdent$
$LA : AttributeList$
$Z : Configuration$
$C, C1 : ConstraintStore$

$[Main1]\ [X_1, Task, [Name, Main]\ LA]\ Z$ **with** $C \Rightarrow X'2\ X'3\ Z$ **with** $C'1$
where $O1 := ()\ [X_1, Task, [Name, Main]\ LA]$
where $X'2 := ()\ [O(2), Task, [Name, FormPrint]\ [Status, all]$
$[NumI, O1.NumI - 1]\ [B, V\_C.Max + 1]\ [E, V\_C.Max + 2]\ natt]$
where $X'3 := ()\ [O(3), Action, [Name, SimplePrint]\ [B, V\_C.Max + 3]\ [E, V\_C.Max + 4]\ natt]$
where $C1 := ()C \uplus [V\_C.Max + 1 \in^? D, V\_C.Max + 2 \in^? D, V\_C.Max + 3 \in^? D,$
$V\_C.Max + 4 \in^? D\ ,\ V\_C.Max + 1 \geq^? O1.B\ \wedge\ V\_C.Max + 3 =^? O1.B\ \wedge$
$V\_C.Max + 2 \leq^? O1.E\ \wedge\ V\_C.Max + 4 \leq^? O1.E\ \wedge\ V\_C.Max + 4 =^? V\_C.Max + 1\ \wedge$
$V\_C.Max + 4 =^? V\_C.Max + 3(+)2$

3.4 The evaluation mechanism

To apply a rule $[\ell]\ \ l \to r$ **where** $p := c$ on a term t (with l and p two syntactic terms), the satisfiability of the condition $p := c$ has to be checked before building the reduced term. Let σ be the matching substitution from l to $t_{|\nu}$. Checking the matching condition $p := c$ consists first of using the rewrite system R to compute a normal form c' of $c\sigma$, when it exists, and then verifying that p matches the ground term c'. If there exists a substitution μ, such that $p\mu = c'$, the composed substitution $\sigma\mu$ is used to build the reduced term $t' = t[r\sigma\mu]_\nu$. Otherwise the application of the rule fails. Note that for usual boolean conditions of the form **if** c, μ is the identity when the normal form of $c\sigma$ is *true*. It may also happen that no normal form is found for $c\sigma$, in which case the rule is said non-terminating.

When the rule is of the form

$$[\ell]\ \ l \to r\ \textbf{where}\ p_1 := c_1 \ldots\ \textbf{where}\ p_n := c_n$$

the matching substitution is successively composed with each matching μ_i from p_i to the normal forms of $c_i\sigma\mu_1 \ldots \mu_{i-1}$, for $i = 1, \ldots, n$. If one of these μ_i does not exist, the application of the rule fails. If no normal form is found at some step i, the rule does not terminate.

When the left-hand side of the rule contains AC function symbols, AC matching is invoked. The term l is said to AC match another term t if there exists a substitution σ such that $l\sigma =_{AC} t$. In general, AC matching can return several solutions, which introduces a need for backtracking for conditional rules: as long as there is a solution to the AC matching problem for which the matching condition is not satisfied, another solution has to be extracted. If the pattern p contains AC function symbols, a general one-to-one AC matching procedure is used [7] to find a substitution μ such that $p\mu = c'$. Only when all solutions have been tried unsuccessfully, the application of this conditional rule fails. When the rule contains a sequence of matching conditions, failing to find a match for the i-th condition causes a backtracking to the previous one.

The evaluation of a generalised matching condition $p_i := (S_i)c_i$ involves the evaluation of c_i and S_i first, and then of the application operator $(\_)\_$. In general, this leads to a multiset of terms. Finally, the pattern p_i is matched with each result in this multiset. If either the multiset is empty, or the matching condition is not satisfied, then the evaluation backtracks to the previous matching condition; otherwise, the evaluation sets a choice point and goes on with one of the returned terms.

Based on this evaluation process, and starting from a query which is a term without variables, ELAN builds a derivation tree in which each branch corresponds to a deduction in rewriting logic. Each node in this tree corresponds to a reachable configuration. Since there is no assumption on termination of rewrite rules, we may get an infinite derivation tree.

Example 6 *The construction of the search tree can be done with two execution modes: a Full mode and a Step-by-Step mode.*

- *The Full mode uses a strategy* `AllTogether` *that develops all branches of the tree, checking at each node that the constraint is satisfiable. We do not compute any solution, but we check at each node that the constraint has at least one solution, so the schedule is feasible. At each leaf, we get a possible decomposition for the main task into primitive actions. The strategy* `AllTogether` *calls the* `Main`, `Form` *and* `Multi` *strategies defined above:*

  ```
  [] AllTogether => Main ; repeat*(Form ; repeat*(Multi)) ; Sat end
  ```

  If we test this example with one simple print task and a complex one with 2 prints and with one simple print task and a complex one with 3 prints, we get 3 schedules for the first test and 8 for the second one.

- *The Step-by-Step mode guides the user during the development of a solution, with a menu presented on the screen:*

  ```
  Could you give us the number associated to the strategy
  you want now to execute (terminated by 'end')?:
  1- Main
  2- FormPrint
  3- MultiPrint
  4- All Results
  5- One Result
  6- Cut this branch
  ```

  The three first choices help the user to develop the tree and to eliminate all complex tasks from the list of tasks. The fourth and the fifth ones guide the application of the COLETTE *strategy: it gives either all results, or only the first one (as for the test of satisfiability). The sixth choice allows the user to cut the current branch of the tree, and the exploration starts again at the last set choice point.*

  The user choice is guided by the display of the current situation, as shown bellow:

  ```
   List of Tasks : <MultiPrint 1 all V_21:V_22>.nil
   List of Actions : <Load V_19:V_20>.<FormKeep V_19:V_22>.
  <Print V_17:V_18>.<Print V_13:V_14>.<Load V_9:V_10>.
  <FormKeep V_9:V_12>.<SimplePrint V_3:V_4>.nil
  ```

  As an example, from the query `Step(<Main 4 split 0:8>)`*, one can reach the following result:*

  ```
  <Print 7:8>.<Load 6:7>.<FormKeep 6:8>.
  <Print 3:4>.<Load 2:3>.<FormKeep 2:4>.
  <Print 1:2>.<Load 0:1>.<FormKeep 0:2>.
  <SimplePrint 4:6>.nil
  ```

  This schedule proposes to execute the simple print task SimplePrint between time 4 and 6. The complex print task is decomposed into three ones that are executed

for the first one between 0 and 2 (Load between 0 and 1 - Print between 1 and 2 - FormKeep between 0 and 2), for the second one between 2 and 4 (Load between 2 and 3 - Print between 3 and 4 - FormKeep between 2 and 4) and for the last one between 6 and 8 (Load between 6 and 7 - Print between 7 and 8 - FormKeep between 6 and 8).

4 Conclusion

Several directions need yet to be further explored.

First, the proposed framework allows us to express concurrency and synchronisation between structured objects. Since we allow several objects in the left-hand sides of rules, this means that in order to apply the rule, these objects needs to match objects simultaneously present in the configuration. More examples need to be studied to illustrate this capability.

Second, we put in our rule formalism the restriction that the rules do not involve constraint variables in their left-hand side. We plan to investigate the consequences of relaxing this restriction.

From the implementation point of view, the translation to ELAN programs must be completely automated. It is currently being implemented.

References

[1] J. Andreoli, U. Borghoff, S. Castellani, R. Pareschi, and G. Teege. Agent Based Decision Support for managing Print Tasks. In *Proceedings of the PAAM'98, London, UK*, 1998.

[2] P. Borovanský. *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France, Oct. 1998. also TR CRIN 98-T-326.

[3] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), Sept. 1996.

[4] C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34:263–293, September 1998.

[5] C. Castro. *Une approche déductive de la résolution de problèmes de satisfaction de contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France, 1998.

[6] F. Durán. *A reflective module algebra with applications to the Maude language.* PhD thesis, University of Malaga, Spain, 1999.

[7] S. Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

[8] B. Liu, J. Jaffar, and H. Yap. Constraint rule-based programming, 1999.